Journal of Systems Architecture 100 (2019) 101661

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

CASS: Criticality-Aware Standby-Sparing for real-time systems

Mingxiong Zhao [a], Di Liu [a,b,*], Xu Jiang [c], Weichen Liu [b], Gang Xue [a], Cheng Xie [a], Yun Yang [a], Zhishan Guo [d]

[a] School of Software, Yunnan University, China
[b] School of Computer Science and Engineering, Nanyang Technological University, Singapore
[c] School of Computer Science and Engineering, University of Electronical Science and Technology of China, China
[d] Department of Electrical and Computer Engineering, University of Central Florida, United States

ARTICLE INFO

Keywords:
Fault tolerance
Real-time systems
Energy-efficiency
DVFS

ABSTRACT

The standby-sparing (SS) is a promising technique which deploys the dual-processor platform, i.e., one primary processor and one spare processor, to achieve fault tolerance for real-time systems. In the existing SS framework, all applications have their backup copies on the spare processor, but, in practice, not all applications on a system are equally important to the system. Some low critical tasks may be traded off for other system objectives. Motivated by this, in this paper, we integrate the concept of criticality into the SS framework. Such integration enables the SS framework to further reduce energy consumption. We propose an offline approach to determine an energy-efficient frequency for the primary processor. Additionally, as the cluster systems are emerging as the mainstream computing platform, we consider the SS technique on the cluster/island systems and propose an algorithm to determine the energy-efficient algorithm for such systems. We evaluate the proposed approach on synthetic tasks and real-platforms. The experimental results demonstrate the effectiveness of our proposed framework in terms of energy efficiency.

1. Introduction

With the rapid development of the transistor technology, a huge amount of transistors are fabricated on a single die to provide high performance. However, as the size of transistors shrinks, the probability that applications suffer from soft error increases [1]. Thus, the reliability issue is arising as another design concern for systems, especially for real-time systems which have a stringent real-time constraints to respect. Different approaches were proposed to ensure fault tolerance for real-time systems, such as [2–6]. Meanwhile, the majority of real-time systems are embedded systems, usually powered by battery, so how to ensure the reliability of the systems in an energy efficient fashion is arising as a new challenge for real-time embedded systems [4–6].

The standby-sparing (SS) technique is a promising approach to ensure fault tolerance by using hardware redundancy while achieving energy efficiency, where the SS architecture has two processors, one *primary* processor and one *spare* processor [5,6]. The execution semantics of the SS is as follows: every task has a main copy on the primary processor which uses dynamic voltage/frequency scaling (DVFS) to save energy, and a backup copy on the spare processor which always executes at maximum frequency for on-time fault recovery. If the main copy completes successfully, i.e., no fault occurred, the corresponding backup copy on the spare processor is canceled to save energy. Otherwise, the backup copy executes to its completion for fault recovery purpose. To effectively balance energy efficiency and fault tolerance, the tasks on the primary processor are scheduled using a real-time scheduling algorithm, such as EDF and RM, but the backup copies on the spare processor are scheduled as late as possible such that they have a high probability to be canceled. More details about SS are discussed in Section 3.2.

All existing SS approaches, like [5,6], treat all tasks equally, i.e., they assume that all tasks are of the same importance to the system and need to be recovered from a fault. However, in practice, not all real-time applications are equally important to a system, and are required to be recovered when a transient error occurs. We can see the evidence from other models and systems, such as the imprecise computational model [7] and the elastic model [8]. Those models either sacrifice quality of service (imprecise model) or performance (throughput in elastic model) to ensure the operation of the whole system under the overloaded situation. Another primary example can be seen in mixed-criticality research [9], where the tasks of higher criticality

* Corresponding author at: School of Software, Yunnan University, China.
E-mail addresses: mx_zhao@ynu.edu.cn (M. Zhao), dliu@ynu.edu.cn (D. Liu), jiangxu@uestc.edu.cn (X. Jiang), liu@ntu.edu.sg (W. Liu), mass@ynu.edu.cn (G. Xue), xiecheng@ynu.edu.cn (C. Xie), yangyun@ynu.edu.cn (Y. Yang), zsguo@ucf.edu (Z. Guo).

https://doi.org/10.1016/j.sysarc.2019.101661
Received 26 May 2019; Received in revised form 20 July 2019; Accepted 5 October 2019
Available online 24 October 2019

can receive extra execution budgets when they overrun their estimated execution time [10,11]. In contrast, the tasks of lower criticality will instead be stopped when they overrun their estimated execution time [12]. The major goal of mixed-criticality systems is to eliminate the system under-utilization while guaranteeing the temporal correctness of all high-criticality tasks. Recently, Brugge in [13] proposed a model which takes into account fault tolerance and different importance of tasks. All those examples mentioned above show the merit of trading off the functionality/performance of lower critical tasks for effective resource usage while guaranteeing the required functionality.

Inspired by this, we in this paper integrate the criticality concept into the SS technique and propose a Criticality-Aware Standby-Sparing (CASS in short) framework for fault-tolerant real-time systems. For the SS framework, if only critical tasks are recovered when a fault happens, i.e., only critical tasks are scheduled on the spare processor, it can reduce the workload on the spare processor, and as a result more backup copies may be canceled on the spare processor. This will lead to a lower operational frequency of the primary processor as well as less energy consumption of the spare processor. Given that many embedded real-time systems are battery-supplied, prolonging the operational time is definitely a preference. For instance, the unmanned vehicle aerials (UVAs) usually have two types of tasks, flight control tasks and multimedia tasks (e.g., video surveillance). It is evident that flight control tasks would be critical to the whole system and needs to be recovered immediately to guarantee the operation of UVA if any fault occurs. Otherwise, it may lead to severe consequences. On the other hand, multimedia tasks may not have to do fault-recovery, because a blurry/missing image/video will not affect the safety of UVA. Then by only ensuring the fault tolerance of flight control tasks, the operational time of the UVA might be extended.

To reduce energy consumption, the SS framework applies DVFS on the primary processor. The existing SS approaches [5,6] make online decisions to scale up/down the frequency for each task. However, for real-time systems, it is preferred to have a guarantee at design time and also DVFS itself occurs some overhead. Frequently changing frequency will thus undermine the effectiveness of DVFS in terms of energy efficiency. In Section 6, on a real-platform, we evaluate a dynamic DVFS approach comparing with a fixed maximum frequency approach in terms of energy efficiency. We see that the dynamic approach does not show advantages over the maximum frequency one. Therefore, we propose an offline approach to determine the frequency for the primary processor, and not only does this approach work for the CASS framework but also for the SS frameworks. The proposed offline approach can be combined with the existing online approaches to achieve better energy efficiency, where the proposed offline approach can be used to determine the frequency in prior at design time and an effective online approach is deployed at runtime to further reduce energy consumption. Additionally, besides the normal SS platform, we first consider the cluster/island platform for the SS framework, where all processors/cores on the same cluster operate at the same frequency. Due to the high hardware overhead [14,15], the increasing number of multicore systems are implemented based on the clusters/islands. Our detailed contributions in this paper are as follows:

- We integrate the criticality concept into the standby-sparing technique for real-time systems. In order to derive an energy-efficient frequency for the SS at design time, we present an analytical approach to compute the overlap between a job of any task and its corresponding backup copy, because the overlap plays a pivot role in determining the offline frequency of the primary processor.
- We analyze the trade-off relationship between the overlap and energy consumption. Based on our analysis, we derive an algorithm, called CASS (Criticality-Aware Standby-Sparing), to determine the energy efficient frequency for the primary processor;
- We consider the cluster/island platform as the SS platform and present an algorithm, called CASS+, to determine offline frequency for the cluster CASS framework.

Evaluation: We use a flight management system (FMS) [10] as our case study and use synthetic task sets to extensively evaluate the effectiveness of our approach in comparison with an online method [5]. This comparison aims to demonstrate the potential of CASS in terms of energy efficiency. In addition, we implement CASS+ on an Intel desktop and use the widely used PARSEC benchmarks [16] to evaluate the effectiveness of CASS+. Experimental results show the effectiveness of CASS and CASS+ in terms of energy efficiency.

The remainder of this paper is organized as follows: Section 2 discusses the related work. Section 3 presents some preliminaries including task model, SS technique and power model. Section 4 gives motivational examples to show the advantage of CASS framework over SS framework. Section 5 presents our analysis framework and proposed algorithms. Finally, Section 6 demonstrates the experimental results and Section 7 concludes this paper and discusses the future work.

2. Related work

In [6], Ejlali et al, introduced the standby-sparing (SS) technique to hard-real-time systems. However, they consider *non-preemptive* scheduling and a task graph with a common deadline. *Non-preemptive* scheduling is known to be NP-hard in the strong sense even for the uniprocessor [17] whereas we consider *preemptive* scheduling. Their approach is inapplicable to periodic tasks which accounts for a big fraction of real-time applications.

In [18], Haque et al. proposed an approach to energy-efficiently use the SS technique for periodic tasks under fixed-priority scheduling. In contrast, in this paper we adopt EDF algorithm [19] which is known to be the optimal algorithm on uniprocessor systems and has a higher utilization bound than fixed-priority scheduling. The work closest to ours is [5], where the same task model and scheduling algorithm are considered. The main difference between ours and [5] is that their approach considers to change operational frequency at the granularity of a task job, i.e., an online/dynamic policy. However, DVFS technique itself occurs considerable overhead [20]. As a consequence, the effectiveness of DVFS may be undermined by frequently varing frequencies. Some experiments on a real-platform show that the dynamic DVFS policy (i.e., changing frequency according to the workload) may not lead to energy efficiency even in comparison with the system always at maximum frequency (see in Section 6). In contrary, our approach will only set a fixed frequency for the processor at design time and this will not cause any overhead at run-time. Additionally, in our work, we integrate the concept of criticality into the SS framework, this means that our approach is more flexible and can leave more space for the primiary processor to scale down its frequency in pursuit of more energy saving.

Our work also has some relevance with energy efficient scheduling and mixed-criticality scheduling [21], but they are not as close as the works discussed above. Thus, we do not discuss them here, and we refer interesting readers to two comprehensive surveys, [22] for energy-efficient real-time scheduling and [9] for mixed-criticality systems.

3. Preliminaries

In this section, we introduce the task model, Standby-Sparing technique, and system model.

3.1. Real-time task model

We consider the implicit-deadline periodic real-time task model that has been widely studied in literature, where there is a task set γ, including n tasks. Each task $\tau_i \in \gamma$ is characterized by $\tau_i = \{T_i, C_i, L_i\}$. T_i denotes *period* and C_i represents *worst-case execution time (WCET)*,

estimated at the maximum frequency $f_{\max}$ of the processor. $L_i = \{True, False\}$ indicates the fault-tolerate requirement. If L_i is true, then it requires a recovery when a fault occurs. Otherwise, L_i is false. In this paper, we consider there are two types of tasks similar to the widely studied dual-criticality systems in mixed-criticality research [9]. For sake of presentation, we call the tasks which need fault recovery FR tasks. The implicit deadline means that the deadline of each task is equal to its period, so we omit deadline parameter in the task specification. With WCET (C_i) and period (T_i), the utilization of each task τ_i is defined as $U_i = \frac{C_i}{T_i}$ and the total utilization of task set γ is $U = \sum_{i=1}^{n} U_i$.

Rational behind L_i: L_i is conceptually similar to criticality levels in mixed-criticality (MC) research [9]. The major goal of mixed-criticality systems is to mitigate the under-utilization of safety-critical systems caused by over-provision on the WCETs of high-criticality tasks. In literature, dual-criticality MC systems have two criticality levels, *high* and *low*. Highly critical tasks have two WCETs, a smaller one for its normal execution and a larger one for its abnormal execution, where highly critical tasks first are scheduled with their smaller WCET and receive extra execution budgets to complete its execution as something abnormal happens, such as recovery from fault [10]. In contrary, low critical tasks only have one WCET and they will not receive any extra budget to complete their execution when any abnormal situation happens to them. This reflects the possibility that not all tasks need to have a fault-tolerate mechanism and motivates us to apply this paradigm to have our CASS framework. The main difference between the well-studied mixed-criticality model and the model considered in this paper is that we do not consider the varing WCET of each task, which is the special feature of mixed-criticality systems.

3.2. Standby-sparing technique

Different from the fault-tolerant appraoches based on the software techniques, such as re-execution/ roll-back [23], the motivation of the SS technique is to utilize hardware-redundancy to guarantee fault-tolerance while reducing energy consumption. As in the literature, the SS technique is mainly used for transient fault. All tasks are scheduled on the primary processor, and the backup copies of tasks are scheduled on the spare processor. If a task completes successfully, then its corresponding backup on the spare processor will be canceled to save energy consumption. Otherwise, the backup copy will be scheduled to its completion to support fault recovery. To guarantee the fast execution of the backup tasks, the spare processor always operates at the maximum frequency. Therefore, the goal of the existing SS approaches strives to minimize the overlap between tasks and their backup copies such that energy consumption of the spare processor can be reduced. However, *as we observe that minimizing overlap strategy applied in [5,6] does not lead to the minimal energy consumption, in some cases, we could trade off the overlap for the lower operational frequency of the primary processor, which compensates the increased energy consumption of the spare processor and thus leads to a more energy efficient system.* An example is given in Section 5 to demonstrate this possible tradeoff. Like other literature considering SS techniques, we only consider the transient fault in this work.

The scheduling algorithms is an important factor for the SS framework to ensure reliability and reduce energy consumption. In this paper, similar to [5], to minimize the overlap, all tasks on the primary processor are scheduled under *earliest deadline first* (EDF) algorithm [19], whereas all backup copies of tasks on the spare processor are scheduled under *earliest deadline as late as possible* (EDL) algorithm [24]. EDF algorithm is known to be optimal on uniprocessor systems and EDL is another version of *earlies deadline* scheduling algorithm [24]. For both algorithms, we use the same tiebreak policy when two or more tasks have the same deadline. The task with a larger period is scheduled first and the task with smaller task ID is given higher priority in the case of having the same period. We select this tiebreak policy for the purpose of reducing the overlap and it does not affect the scheduling performance on uniprocessor systems [25].

3.3. System model

For power consumption of processors, we deploy the widely used power model [15,22,26]

$$P = af^b + s, \tag{1}$$

where a and $b \in [2, 3]$ are both hardware-related parameters. The first part of Eq. (1) denotes the dynamic power consumption which accounts for power consumption due to execution activity and is frequency-related. s denotes the leakage/static power which is not related to frequency and is consumed as soon as the processor is switched on.

It is known that due to the presence of leakage power consumption it is not beneficial to scale down the operational frequency below a certain frequency level, because it may lead to even more energy consumption. Such frequency level is called *critical frequency*, denoted as f_{crit}.

With Eq. (1), we can compute energy consumption of a task set within a hyper-period. Hyper-period is the *least common multiple* of periods of all tasks, denoted as *hp*. Since every hyper-period repeats the same schedule, we compute energy consumption of task set γ over one hyper-period. The total energy consumption is the summation of energy consumption of the primary processor and the spare processor.

$$E = E_p + E_s \tag{2}$$

where E_p and E_s are energy consumption of the primary processor and the spare processor, respectively. They are computed as follows:

$$\begin{aligned} E_p &= hp\left(a \times f_k^b \times \frac{\sum_{\forall \tau_i \in \gamma} C_i/T_i}{f_k/f_{\max}} + s\right) \\ &= hp(a \times f_{\max} \times f_k^{b-1} \times U + s) \end{aligned} \tag{3}$$

$$E_s = a \times f_{\max}^a \times \sum_{\forall \tau_j \in \gamma_{FT}} \sum_{k}^{hp/T_j} O_{j,k}^o + s \times hp \tag{4}$$

γ_{FT} is the set of all FR tasks. $\sum_{\forall \tau_j \in \gamma_{FT}} \sum_{k}^{hp/T_j} O_{j,k}^o$ denotes the total overlap of *FR* tasks within a hyper-period.

4. Motivational example

In this section, we present two motivational examples to show the advantages of the CASS framework over the SS framework in terms of energy efficiency.

4.1. Real-life benchmarks

We implement the SS technique on Ubuntu 16.04 of a desktop with Intel i7-8700 CPU of 6 cores and 16GB memory and the maximum frequency of each core is 3.2 GhZ. Likwid-powermeter tool [27] is used to measure the energy consumption from the experimental desktop. CPU1 is used as the primary processor and CPU2 is used as the spare processor. In addition, to prevent the interference on the primary or spare processor, we fix the cpu affinity of likwid-powermeter to CPU3. We select two benchmarks, freqmine and facesim from the widely used PARSEC benchmark suite [16] for the illustrative purpose, where both benchmarks are compiled with *'gcc-serial'*, i.e., a single thread executable is generated. Then, the benchmarks are executed with *'simlarge'* input data. More details about compilation and execution of PARSEC can be found in [16]. WCETs are estimated according to the real measurement on the experimental platform with some additional overhead. The WCETs of freqmine and facesim are 5 s and 27 s, respectively, and we assign a period of 30s and 50s to them respectively. The parameters of these two benchmarks are given in Table 1. The total utilization of these

Table 1
Benchmark parameters.

Benchmark	T	C	L
Freqmine	30 s	5 s	True
Facesim	50 s	27 s	False

Table 2
Experimental results of the motivational example.

	Approach	Energy	Diff
Dynamic	SS	2420.29J	604.92J
	CASS	1815.37J	25%
Max	SS	2398.49J	556.94J
	CASS	1841.55J	23%

Table 3
Motivational Example.

task	C	T	L
A	30 ms	50 ms	False
B	20 ms	100 ms	True

two applications is 70.6%. In addition, we set benchmark facesim's L to False, i.e., in the CASS framework, benchmark facesim is not executed on the spare core.

We measure the energy consumption of both approaches for an interval of 150 s which is the hyperperiod of the two benchmarks. In this example, we test two DVFS mechanisms, one is the fixed frequency approach (considered in this paper), denoted as 'Max' in Table 2, where we fix the system's frequency to the maximum frequency. Another method considered is a 'dynamic' approach, i.e., the frequency changes according to the utilization, similar to the approach presented in [18]. In this example, we deploy Ubuntu's default 'ondemand' policy to emulate this online approach [28]. Experimental results are summarized in Table 2, where SS and CASS denotes the traditional SS technique and the criticality aware SS proposed in this paper, respectively.

We see from the results that if only FR task freqmine is scheduled on the spare processor, energy consumption can be reduced by 604.92 Joule and 556.94 Joule, for 'dynamic' and 'Max' approach, respectively. That is more than 20% energy saving. This shows a significant reduction on energy consumption. Moreover, we see that even executing at the maximum frequency does not consume a lot more energy than the dynamic approach, even in no criticality case, the fixed frequency consumes less energy than the dynamic case. This inspires us to find an offline approach to determine an energy-efficient and fixed frequency for the CASS framework.

4.2. Illustrative example

The real platform example shows the potential of the CASS technique in terms of energy consumption. However, due to the limitation of the new Intel CPU, under 'userspace' mode, all cores' frequencies are set as the same. Thus, it cannot show how the CASS save energy consumption by scaling the primary processor's frequency. By only scheduling FR tasks on the spare processor, the system is able to cancel more backup copies and leaves more room for the primary processor to scale down its frequency, thereby reducing more energy consumption. In this section, we use an illustrative example to show this.

Given a task set shown in Table 3, there are two tasks where: task A is low critical and needs no fault recovery, while task B is a critical task and thus has a backup copy upon the spare processor. Power parameters and frequency levels used in this example are from an ARM Cortex A15 core measured in [26]. Note that as indicated in [15], the critical frequency f_{crit} of Cortex A15 is not observed within range [1200 MhZ, 2000 MhZ],

Table 4
Power parameters of ARM cortex A15 from Liu et al. [26].

$b(W/MhZ^2)$	a	$P_s(W)$
3.03×10^{-9}	2.621	0.155
frequency levels (MhZ)	1200 1400 1600 1800 2000	

this means that the operational frequency can be scaled down as much as possible once the deadlines can be guaranteed.

First, we show how tasks executes in the SS framework, i.e., all tasks have backup copies on the spare processor. In this case, to save energy consumption, the objective is to minimize the overlap between tasks on the primary processor and their corresponding backups on the spare processor such that the execution of backup copies can be canceled as early as possible. In this example, the schedule of task A and B within a hyper-period 100 ms derived by using the existing approach [5] is given in Fig. 1, where "Overlap" denotes the overlap execution on the spare processor when there is no fault. It can be seen that there exists overlap even as the primary processor operates at the maximum frequency of $2GhZ$, so the operational frequency of the primary processor will not be scaled down. The total overlap is 20 ms.

In contrast, if we only have the FR task on the spare processor shown in Fig. 1, the advantage is evident. Since there is no overlap and the spare processor does not consume dynamic power, the primary processor can scale down its frequency to 1.6 GhZ while not violating deadline constraints. Energy consumption is reduced due to the lower frequency. Energy consumption of the two approaches is summarized in Table 5. We see that CASS saves energy consumption over SS by 36%. This example shows how the CASS framework utilizes the advantages of fewer FR tasks on the spare processors to scale down the frequency of the primary processors for better energy efficiency.

5. The proposed approach

The preceding section shows the benefit of the CASS framework. In this section, we address the key issue in this framework, i.e., determining a proper frequency for the primary processor.

The frequency selection of the primary processor is determined by two factors: (1) the total utilization of the primary processor: on uniprocessor systems, the processor can scale down its frequency as long as the total utilization is smaller than or equal to 1 under EDF scheduling. Thus, the total utilization plays an essential role in determining the frequency; and (2) the total overlap of FR tasks: the previous approaches [6,18] always strive to minimize the overlap between tasks and their backup copies. However, in some cases, increasing the overlap on the spare processor may have the primary processor scaled down its frequency further which as a result saves energy consumption of the whole system. Let us see from the following example.

Example 1. Suppose that we have a task with $C = 25$s (obtained at maximum frequency) and $T = 50$s. It is not difficult to see that the main job on the primary processor and the spare job on the spare processor should not have any overlap if they execute at the maximum frequency under EDF and EDL scheduling algorithms, respectively, but any lower frequencies will generate overlap. Therefore, according to the conventional rule of the SS framework, the frequency of the primary processor then should be fixed at the maximum frequency. In this case, considering the power parameters shown in Table 4, the energy consumption in this scenario is 34.0 mJ, here we only compute the dyanmic power consumption because the static power consumption will be the same for both cases. However, if the operational frequency of the primary processor is scaled down from 2000 MhZ to 1600 Mhz. The overlap is now 6s, but the energy consumption is 31.6 mJ which is less than the minimal overlap case. Details about this example is given in Table 6.

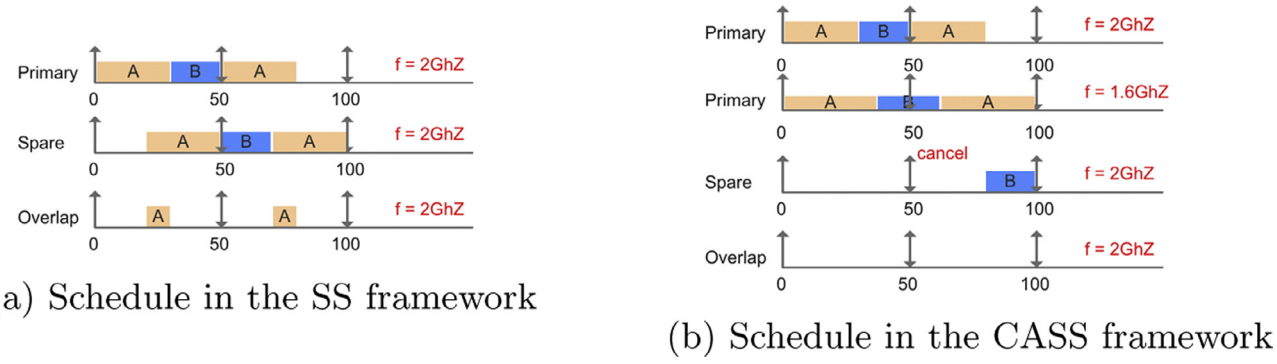


Fig. 1. Motivational example.

Table 5
Energy comparison between two approaches.

	Primary	Spare	Total
SS	124.3 mJ	42.7 mJ	167mJ
CASS	91.2mJ	15.5mJ	106.7mJ

Table 6
The example shows that the minimal overlap does not lead to the least energy consumption in some cases.

Scenario	f of Primary	Overlap	E_p	E_s	Total
Minimal overlap	2000 MhZ	0	34.0 mJ	0	34.0 mJ
Not minimal	1600 MhZ	6	23.6 mJ	8 mJ	31.6 mJ

From Example 1, it is seen that minimizing the overlap does not lead to the minimal energy consumption. To determine the most energy-efficient frequency based on the overlap, we need to precisely compute the overlap of each task such that the desired frequency can be determined. In addition, in contrast to the previous approaches, our approach is an offline approach which can avoid the frequency scaling overhead while guaranteing the hard real-time constraints.

To compute the overlap, we need to know the finish time and start time of a task on the primary and the spare processor, respectively.

5.1. Finish time on the primary processor

The primary processor uses EDF scheduling to schedule tasks and EDF scheduling is a dynamic-priority and work-conserving scheduling algorithm. The work-conserving property of a real-time scheduling algorithm means that the processor cannot be idle if there is a task pending for execution. When we have a closed system, i.e., all tasks are known in prior to the system design, then the schedule table of all tasks under EDF algorithm can be constructed [29] and as results the finish time of each job in this schedule can be known. However, there is no existing way to compute the finish time, so in this section we present an approach.

To compute the finish time of a job under EDF scheduling, except the given parameters of all tasks, we also need to know the idle interval in the schedule. In [24], Chetto and Chetto proposed an iterative approach to compute idle intervals within a hyper-period. Their approach first computes an arrival time vector $\mathcal{E} = \{e_0, e_1, \dots, e_{hp}\}$ with $e_x < e_{x+1}$, $e_0 = 0$ and $e_{hp} = hp$, where $e_x \in \mathcal{E}$ denotes a distinct arrival time from task set γ. The approach to compute the idle interval is not presented here, and interesting readers are refered to [24]. We use $\Omega(e_x)$ to denote the total length of idle intervals before e_x. With the computed idle time, we can compute the finish time of a job under EDF scheduling

Lemma 1. *Using EDF scheduling, given a task set γ, one task $\tau_i \in \gamma$ and one absolute deadline $e_x \in \mathcal{E}$, the finish time of the job of τ_i scheduled with deadline at e_x is computed by the following:*

$$F_i(e_x) = \sum_{j=1}^{n} \left\lfloor \frac{e_x}{T_j} \right\rfloor C_j - \left(\sum_{\forall \tau_m \in \hat{\gamma}(e_x)} C_m \right) + \Omega(e_x - T_i) \tag{5}$$

where $\hat{\gamma}(e_i)$ denotes all tasks which have an absolute deadline at e_x but a smaller period or a larger task id when they have the same period as task τ_i.

Proof. Let J_i denote the job of task τ_i with deadline at e_x. First, $\lfloor e_x/T_j \rfloor$ computes how many jobs task τ_j has completed before e_x. Thus, $\sum_{j=1}^{n} \lfloor e_x/T_j \rfloor C_j$ denotes the total workload with deadline before or at e_x. However, it may happen that multiple tasks have the same deadline at e_x. Then with the tiebreaker policy explained in Section 3.2, tasks pertaining to $\hat{\gamma}(e_i)$ must be scheduled after the job of task τ_j, so we need to remove these workload, i.e., $\sum_{\forall m \in \hat{\gamma}(e_x)} C_m$. Moreover, idle times affect the finish time of job J_i as well. Due to the work-conserving property, the processor cannot be idle if there is a task pending. If a job is released, it must proceed to its completion without idle. Thus, we just need to consider the idle time happened before the release of job J_i. Since we consider the implicit deadline task model, $e_x - T_i$ denotes the release time of job J_i and $\Omega(e_x - T_i)$ denotes the idle time occurs before e_x.

Taking into account all these, by using Eq. (5), the finish time of job J_i of task τ_i with deadline at e_x can be obtained. □

If the idle times and the arrival times of a task set is known, all finish times of all tasks within a hyper-period under EDF scheduling can be computed by using Lemma 1.

5.2. Start time on the spare processor

Lemma 1 helps us determine the finish times of all tasks within a hyper-period on the primary processor under EDF scheduling. To compute the overlap of a task, we also need to know the start time of the corresponding task on the spare processor. In this section, we present the way to compute the start time of a task under EDL algorithm.

In EDL scheduling, all jobs are dispatched as late as possible. Thus, as indicated in [24], different from EDF scheduling, idle times in EDL scheduling immediately follow arrival times. Taking into account this fact and the feature of EDL scheduling, we can compute the start time of jobs under EDL scheduling by the following lemma:

Lemma 2. *Using EDL scheduling, given a task set γ, a task $\tau_i \in \gamma$ and one absolute deadline $e_x \in \mathcal{E}$, the latest start time of the job of τ_i with deadline at e_x is computed by the following:*

$$R_i(e_x) = \Omega(e_x) + \sum_{j=1}^{n} \left\lfloor \frac{e_x}{T_j} \right\rfloor C_j - \sum_{\forall \tau_m \in \hat{\gamma}(e_x)} C_m \tag{6}$$

where $\hat{\gamma}(e_x)$ denotes all tasks which have an absolute deadline at e_x but a smaller period or a larger task id when they have the same period as task τ_i.

Proof. The proof is similar to the one of Lemma 1. The main difference comes from idle times $\Omega(e_x)$. In EDL scheduling, idle times immediately

follow arrival times. Therefore, to compute the start time of τ_i with deadline e_x, all idle times before e_x should be taken into account. □

If the idle times and the arrival times of a task set is known, all start times of all tasks within a hyper-period under EDL scheduling can be computed by using Lemma 2.

5.3. Overlap

With Lemmas 1 and 2, we can compute the overlap between any job of a task on the primary processor and its corresponding backup copy on the spare processor.

Theorem 1. *Given a task set γ and a job of task τ_i with deadline at e_x, the overlap between the job on the primary processor and its corresponding backup copy on the spare processor can be computed as follows:*

$$O_i(e_x) = F_i(e_x) - R_i(e_x) \tag{7}$$

where

$$O_i(e_x) = \begin{cases} \leq 0 & \text{no overlap} \\ > 0 & \text{there is overlap} \end{cases}$$

Proof. It is evident from Lemmas 1 and 2. □

5.4. Energy-efficient frequency

By using Theorem 1, we could analytically compute the overlap for the CASS systems. In previous literature, the main objective of the SS framework is to *minimize* the overlap between the main job and its backup. However, as we showed in Example 1, it is seen that actually minimizing overlap does not always lead to the minimal energy consumption. In some cases, overlaps can be traded off for lower energy consumption. Then, a naive approach to find the energy efficient frequency could try the all possible frequencies, and everytime compute the new overlap. However, this approach is computationally expensive if the hyperperiod is large and there are many tasks in a taskset [30]. In this section, based on Theorem 1, we present a simple way to check whether scaling down the frequency of the primary processor leads to lower energy consumption for the whole system. For compactness of presentation, let $[\![I]\!]_0 = \max(0, I)$. $[\![I]\!]_0$ is used to determine the overlap length for computing energy consumption. When $I \leq 0$ and $[\![I]\!]_0 = 0$, there is no overlap. On the other hand, if $I > 0$ and $[\![I]\!]_0 = I$, there is an overlap consuming energy.

Proposition 1. *Given a task set γ, f_n leads to more energy-efficiency if the following holds*

$$\left(\frac{f_n}{f_{max}}\right)^{a-1} < 1 + \frac{[\![O_{max}]\!]_0 - [\![O_n]\!]_0}{U_\gamma \times hp} \tag{8}$$

where $O_{\max}$ denotes the total overlap when the primary processor executes at maximum frequency. O_n denotes the total overlap obtained at frequency f_n

Proof. If the primary processor executes at maximum frequency, it has:

$$E_{max} = a \cdot f_{max}^b \cdot U \cdot hp + a \cdot f_{max}^b \cdot [\![O_{max}]\!]_0$$

Here, we omit the static power consumption, because they remain the same even if we change the frequency.

If the primary processor scales down its frequency to f_n ($< f_{\max}$), it has:

$$E_n = a \cdot f_n^b \cdot \frac{U \cdot hp}{f_n/f_{max}} + a \cdot f_{max}^b \cdot [\![O_n]\!]_0$$

where O_n ($\geq O_{\max}$) denotes the new overlap. Then, if $E_n < E_{\max}$, frequency f_n is more energy efficient.

$$\begin{aligned} & a \cdot f_n^b \cdot \frac{U_\gamma \cdot hp}{f_n/f_{max}} + a \cdot f_{max}^b \cdot [\![O_n]\!]_0 \\ & < a \cdot f_{max}^b \cdot U_\gamma \cdot hp + a \cdot f_{max}^b \cdot [\![O_{max}]\!]_0 \end{aligned}$$

(By removing a)

$$\begin{aligned} & \Leftrightarrow f_n^b \frac{U_\gamma \cdot hp}{f_n/f_{max}} + f_{max}^b [\![O_n]\!]_0 \\ & < f_{max}^b \cdot U_\gamma \cdot hp + f_{max}^b [\![O_{max}]\!]_0 \end{aligned}$$

(By dividing $f_{\max}$ at both sides)

$$\begin{aligned} & \Leftrightarrow f_n^{b-1} U_\gamma \cdot hp + f_{max}^{b-1} [\![O_n]\!]_0 \\ & < f_{max}^{b-1} \cdot U_\gamma \cdot hp + f_{max}^{b-1} [\![O_{max}]\!]_0 \end{aligned}$$

$$\Leftrightarrow f_n^{b-1} U_\gamma \cdot hp < f_{max}^{b-1}(U_\gamma \cdot hp + [\![O_{max}]\!]_0 - [\![O_n]\!]_0)$$

(Since $U_\gamma \cdot hp$ and $f_{\max}^{b-1}$ are positive)

$$\Leftrightarrow \left(\frac{f_n}{f_{max}}\right)^{b-1} < 1 + \frac{[\![O_{max}]\!]_0 - [\![O_n]\!]_0}{U_\gamma \times hp}$$

□

Proposition 1 provides a simple way to determine whether a frequency is energy-efficient when comparing to the maximum frequency, but we still need to compute the new overlap at the new frequency everytime. On the SS platform, the frequency of the spare processor is fixed at the maximum frequency, and this means that the start times of backup copies are fixed. The frequency variation of the primary processor only affect the finish times of the tasks. Therefore, we can use the following proposition to compute the new overlap based on the overlap obtained at the maximum frequency.

Proposition 2. *Given the total overlap $O_{\max}$ obtained at maximum frequency, if the primary processor could be scaled down to frequency f_n without violating deadline constraints, the total overlap O_n at frequency f_n can be computed as follows:*

$$O_n = U \cdot hp\left(\frac{f_{max}}{f_n} - 1\right) + O_{max} \tag{9}$$

Proof. Since the backup jobs on the spare processor are always executed at the maximum frequency, their starting time under EDL scheduling algorithm is fixed. On the primary processor, frequency scaling does not change tasks' scheduling order due to the unchanged deadline, but it does delay the finish time of each task. $U \cdot hp(\frac{f_{\max}}{f_n} - 1)$ denotes the total increased workload by scaling down the operational frequency to f_n on the primary processor. The scaling-down frequency leads to a longer overlap which are caused by the increased workload. Thus, the new overlap under frequency f_n can be computed by Eq. (9). □

With Proposition 2, we analyze the different cases in Proposition 1.

- $O_{\max} < 0$ and $O_n \leq 0$: This case denotes that even scaling down the frequency to f_n does not generate overlap, then in this case we scale down frequency as low as possible once the frequency can guarantee the deadline constraints.
- $O_{\max} < 0$ and $O_n > 0$: This case denotes that scaling down frequency to f_n generates some overlap, then in this case we have

$$\begin{aligned} & \left(\frac{f_n}{f_{max}}\right)^{b-1} < 1 + \frac{[\![O_{max}]\!]_0 - [\![O_n]\!]_0}{U_\gamma \times hp} \\ & \Leftrightarrow \left(\frac{f_n}{f_{max}}\right)^{b-1} < 1 - \frac{O_n}{U_\gamma \times hp} \\ & \Leftrightarrow \left(\frac{f_n}{f_{max}}\right)^{b-1} < 1 - \frac{U \cdot hp(\frac{f_{max}}{f_n} - 1) + O_{max}}{U_\gamma \times hp} \\ & \Leftrightarrow \left(\frac{f_n}{f_{max}}\right)^{b-1} < 2 - \frac{f_{max}}{f_n} - \frac{O_{max}}{U_\gamma \times hp} \end{aligned} \tag{10}$$

- $O_{\max} \geq 0$ and $O_n > 0$: This case denotes that before scaling down, there already exists overlap, then we have

$$\left(\frac{f_n}{f_{max}}\right)^{b-1} < 1 + \frac{[\![O_{max}]\!]_0 - [\![O_n]\!]_0}{U_\gamma \times hp}$$
$$\Leftrightarrow \left(\frac{f_n}{f_{max}}\right)^{b-1} < 1 + \frac{O_{max} - O_n}{U_\gamma \times hp}$$
$$\Leftrightarrow \left(\frac{f_n}{f_{max}}\right)^{b-1} < 1 + \frac{O_{max} - U \cdot hp(\frac{f_{max}}{f_n} - 1) - O_{max}}{U_\gamma \times hp}$$
$$\Leftrightarrow \left(\frac{f_n}{f_{max}}\right)^{b-1} < 2 - \frac{f_{max}}{f_n} \quad (11)$$

From the above analysis, if we know $O_{\max}$, it is easy to check whether the frequency satisifies the metric according to different cases.

5.5. Proposed algorithm

With the analysis given in the previous section, we present an algorithm to determine the proper frequency for the primary processor, namely Criticality-Aware Standby-Sparing (CASS).

The pseudo code of CASS is presented in Algorithm 1. CASS simply uses the analysis results obtained from Proposition 1 and takes the frequency as the operational frequency which satisifies Proposition 1 and causes the biggest difference between two sides of inequality (8). We select the frequency which can have the biggest difference at the two sides, because it leads to the most energy saving.

Algorithm 1: CASS: Determine the operational frequency for the primary processor.

input : task set γ and frequency set in decreasing order
output: the operational frequency f_o of the primary processor

```
1  f_o ← f_m
2  Compute O_max using Theorem 1
3  F = Find all frequencies which are higher than critical frequency
   f_crit and satisify Proposition 1.
4  f_o = frequency which have the biggest difference between the
   left-hand sideand the right-hand side of Proposition 1.
5  return f_o
```

Complexity analysis: The complexity of CASS is determined by the complexity of computing $O_{\max}$ or the frequency levels a processor supports. Computing the $O_{\max}$ is dependent on the number of tasks and the length of hyperperiod. When it comes to the hyper-period, the computational complexity in the worst-case is pseudo-polynomial [31], so Algorithm 1 has a pseduo-polynomial complexity in the worst-case.

5.6. CASS on cluster systems

Nowadays, the increasingly number of hardware systems are implemented as a cluster/island system, i.e., several processors are on the same voltage/frequency cluster/island to reduce the hardware overhead caused by the per-core power/frequency regulator [14,15], and the processors on the same cluster/island operates at the same frequency. The traditional SS technique always assumes that the platform supports the per-core DVFS and the spare processor always executes at the maximum frequency, but as we show in Example 1 the minimum overlap does not necessarily lead to the minimum energy consumption. This observation may be also applicable in the cluster system, and thus scaling down the frequency of the spare processor may still have an energy-efficient system. In this section, we investigate how to determine an enery-efficient frequency for the CASS where both primary and spare processors are on the same cluster.

On the per-core DVFS systems, we can have the primary processor operate at a proper frequency for energy efficiency and the spare processor always at the maximum frequency. Thus, as soon as the tasks scheduled on the spare processor is known, their start times are fixed. Then, scaling down the frequency of the primary processor just causes the delayed finish times of tasks on the primary processor, so we can compute the new overlap according to the overlap obtained at the maximum frequency and the new frequency (Proposition 2).

However, for the cluster systems, the start times of tasks on the spare processor are changed along with the scaling-down of the primary processor, so both start times and finish times need to be recalculate. Therefore, Proposition 1 is not applicable any more. In this case We present a new algorithm, namely CASS+, for the CASS on the cluster system.

The pseudo code of CASS+ is presented in Algorithm 2 . The fundamental concept behind this algorithm is to check whether the increased overlap will lead to lower energy consumption of the whole system. It starts with the maximum frequency, computes the energy consumption of the system, and gradually scales down the operational frequency to see whether energy consumption reduces. Then, it returns the frequency having the least energy consumption as the operational frequency of the cluster system. Note that we only consider the frequency level which is greater than critical frequency f_{crit}. If it finds a frequency leads to higher energy consumption, it will skip all frequencies lower than this one.

Algorithm 2: CASS+: Determine the operational frequency for the cluster system.

input : task set γ and frequency set in decreasing order
output: the operational frequency f_o of the cluster system

```
1  f_o ← f_m
2  E_cur = 0
3  for f_k ∈ [f_m, f_{m-1}, ..., f_crit] do
4      if U > f_k/f_m then
5          break
6      E_p(f_k) ← Using Eq. (3) to compute energy consumption of the
       primary processor at frequency f_k
7      for ∀τ_j ∈ γ_FR do
8          Compute the start times of task τ_i within a hyper-period
9          for ∀e_x ∈ E_j do
10             O_k+ = [[O(e_x, f_k)]]_0 (using Eq. (7) to computeO_j(e_x)
               while the operational frequency is f_k)
11     Compute E_p(f_k) and E_s(f_k)
12     if E_cur == 0  ||  E_cur > E_p + E_s then
13         E_cur = E_p + E_s
14         f_o ← f_k
15     else
16         Break
17 return f_o
```

Complexity analysis: Usually, for a processor, the frequency levels it supports are very limited, and let m denote the number of frequency levels greater than f_{crit}. For the second for-loop, the maximum size of γ_{FR} is equal to the size of task set γ. However, for the third for-loop, it depends on the hyper-period, as shown in previous work [31], the computational complexity in the worst-case is pseudo-polynomial. Therefore, in the worst-case, Algorithm 2 has pseudo-polynomial complexity.

6. Evaluation

In this section, we evaluate our proposed approaches CASS and CASS+ in terms of energy-efficiency. This evaluation is mainly to show the potential of the CASS framework in terms of energy efficiency. It can be considered as an effort toward more energy efficiency in the SS

Table 7
Parameters of FMS application.

τ	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
T	5000	200	1000	1600	100	1000
C	20	20	20	20	20	20
FR	True	True	True	True	True	True
τ	τ_7	τ_8	τ_9	τ_{10}	τ_{11}	
T	1000	1000	1000	1000	1000	
C	20	100	100	100	100	
FR	True	False	False	False	False	

framework by trading off the fault tolerant capability of some low critical tasks.

This evaluation includes three parts, a case study, synthetic task evaluation and real-platform evaluation. In the case study, we use a Flight Management System (FMS) application which has been used in the research of mixed-criticality systems [32]. Then, we extensively evaluate the effectiveness of our approach by using synthetic task sets. For these evaluations, we use the power parameters given in Table 4 of Section 4, and additionally we take an existing online approach [5], namely CCSPT for comparison. This comparison is to demonstrate the potential of the CASS framework in terms of energy efficiency. Some existing offline approach can be combined with our proposed approach to achieve better energy efficiency. We compare our approach with the reference approach with different parameter settings, detailed in Section 6.2. The metric used is the energy saving from our approach, which is computed by the following equation,

$$Saving = \frac{E_{ref} - E_{our}}{E_{ref}} \times 100\%$$

6.1. Case study

FMS application is an avionic use-case, which conforms to the standard of DO-178B in avionic industry and is a typical example of the system with different critical applications [10]. This case study only consists of a subset of original tasks in FMS, and all task parameters are given in Table 7. FMS has two types of tasks, high or low criticality. In [10], only high-criticality tasks have the fault-recovery mechanism, (i.e., re-execution), whereas low-criticality tasks do not recover from fault. This is similar to the scenario we consider, so we map high-criticality tasks to FR tasks and low-criticality tasks to normal tasks. Note that in [32], they only gave a range of WCET for each task. In this evaluation, for FR tasks, we select the upper bound of the range as their WCET, and for normal tasks we select a mean value from the range as their WCETs.

We compare our approach with CCSPT in terms of energy efficiency. CCSPT uses a per-task frequency scaling and decides the operational frequency based on the slack which the system has upon the task arrival. However, frequency scaling occurs considerable overhead [20], as we measured on ODROID XU3 board, the scaling-up and scaling-down of Cortex A15 core costs at most 40 ms and 60 ms[1], respectively. This may significantly affect the applicability of CCSPT, so, in this comparison, we consider one reference approach without any overhead and another reference approach with overhead of 40 ms. In order to guarantee the schedulability of tasks, we slightly modify CCSPT for the second reference approach to ensure the real-time guarantee will not be violated when scaling frequency with overhead.

Moreover, since CCSPT also utilizes the dynamic slack generated by tasks which finish earlier than their WCETs. To evaluate this scenario, we assign a fixed slack ratio to all tasks, assuming that the real execution times of each task are equal to $C_i \times (1 - \text{slack ratio})$. Then based on the real execution time, CCSPT computes the frequency for each task. *Note that we consider the slack scenario with frequency scaling overhead.*

[1] Note that this overhead is measured based on the cluster architecture,because ODROID XU-3 only supports cluster frequency scaling. In addition, our timestamp measurement also occurs some overhead.

Table 8
Experimental results of FMS.

Different Reference approaches	Energy Saving
CCSPT	30.3%
S0_O	30.3%
S10_O	22.6%
S20_O	12.9%
S30_O	0.5%

The experimental results are summarized in Table 8, where CCSPT denotes the original CCSPT approach without overhead and SX_O denotes a setting with slack ratio X% and scaling overhead. In this experiment, we consider four slack ratios {0, 10%, 20%, 30%}. Comparing to CCSPT, the advantage of our approach is that we have fewer backup copies on the spare processor, thus leading to a lower frequency of the primary processor with lower energy consumption. When there is no slack, our approach can save energy consumption by 30% regardless of the presence of scaling overhead (CCSPT and S0_O). The rational behind is that the task set has relatively high utilization of 0.78, so CCSPT will not scale its frequency quite often with the objective of minimizing overlap. Therefore, we see that even taking into account the overhead, the energy saving is almost the same. Then, in our approach, because of fewer backup copies on the spare processor and a better objective (not minimizing overlap), our approach achieves a great energy saving. We see that with the increased slack ratio, the energy saving from our approach reduces as well, because for CCSPT they find more space to do scaling for each job. This trend is very intuitive, but it is worth *noting that our approach always takes the worst case into account. Our approach also can deploy some on-line slack reclamation technique, but we leave it for the future consideration.*

6.2. Synthetic task sets

To extensively evaluate the effectiveness of our approach, we vary the task parameters to generate diverse synthetic task sets. The task generator is based on the widely-used *UUnifast* [33] which is used to generate unbiased task utilization. It takes as inputs the number of tasks n and the total utilization U and generates individual utilization u_i for n tasks. The generation procedure is summarized as follows:

- For each task, utilization u_i is generated using *UUnifast*;
- Period T_i is uniformly generated from range [100 ms, 1000 ms];
- C_i is computed as $C_i = u_i \cdot T_i$; and
- Let $pc \in (0, 1)$ denote the possibility whether the generated task is a FR task.

In this experiment, we vary the total utilization U, the task number and the pc value to generate task sets with different parameter settings. This task generation configuration is widely used in the evaluation of mixed-criticality scheduling algorithms [9]. One experimental results are summarized in Figs. 2–4.

In each figure, we vary U from 0.5 to 0.95 with a step of 0.05 and generate 1000 task sets for each given U. The task number and pc vary. We consider four reference approaches similar to those in the case study, and the slack ratio is selected from {0, 10%, 20%}. From Fig. 2, we see:

- the line of CCSPT overlaps the one of S0_O, because the objective of CCSPT is to minimize the overlap between the main task and its backup, so when there is no slack it significantly limits the possibility to scale down the frequency. Since the frequency does not change frequently, no scaling overhead affect results;
- when U increases to 0.8, the saving from CCSPT and S0_O do not decrease too much after that. This is because that when utilization

Table 9
Selected PARSEC benchmarks and their measured execution times.

Benchmark	blackscholes	bodytrack	facesim	ferret
Measured execution time	3s	10s	27s	12s
Benchmark	freqmine	streamcluster	vips	x264
Measured execution time	5s	14s	8s	20s

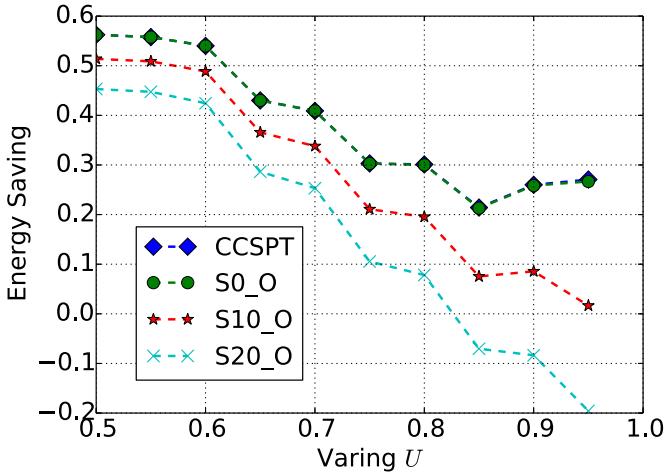


Fig. 2. Task number 7 and $pc = 0.5$.

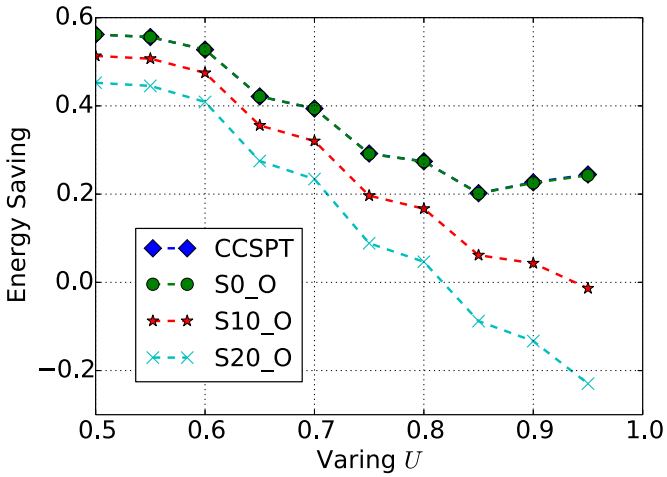


Fig. 4. Task number 7 and $pc = 0.7$.

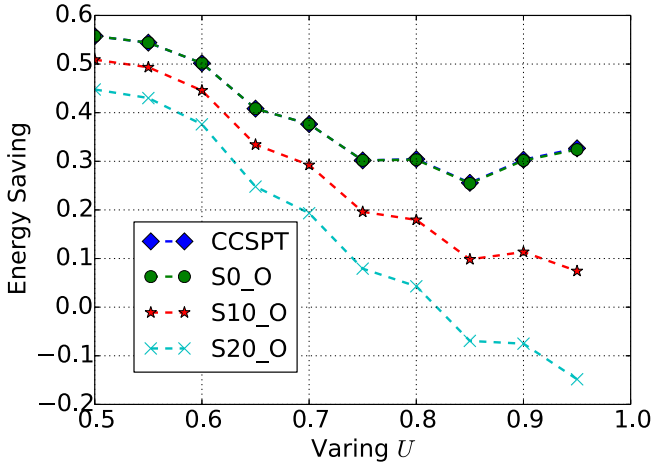


Fig. 3. Task number 5 and $pc = 0.5$.

U is large, our approach also cannot scale down frequency further. Therefore, the energy saving remains.

- when the utilization is greater than 0.8, we see that S20_O actually consumes less energy than our approach. Because our approach cannot scale down frequency on primary processor in this case, whereas a 20% slack creates more space for S20_O to scale down frequency (Fig. 3).

All experimental results show the similar trend as Fig. 2. However, we find that the increased number of tasks in a task set reduces the energy saving from our approach shown in Fig. 4. The rational behind is that when the number of tasks increases, it is less likely to have tasks with large utilization. Our approach may benefit from normal tasks with a large utilization. On the other hand, when increasing pc, we may have more FR tasks in a task set. It is not difficult to think that more FR tasks will mitigate the effectiveness of the CASS technique.

6.3. Real-platform

The case study and the synthetic task sets evaluate the effectiveness of the CASS framework in terms of energy efficiency. In this section, we evaluate CASS+ on a real platform with real benchmarks. The platform we used is a desktop with Intel i7-8700 CPU of 6 cores and 16GB memory and the maximum frequency of each core is 3.2GhZ and the operating system is Ubuntu 16.04 with kernel version 4.15.0. We schedule PARSEC benchmark suite [16] on the experimental platform, but we only select several representative benchmarks (they have different execution time). All benchmarks are compiled with *'gcc-serial'*, i.e., a single thread executable is generated and are fed with *'simlarge'* input data. The selected benchmarks and their measured execution times are given in Table 9. Likwid-powermeter tool [27] is used to measure the energy consumption from the experimental desktop. CPU1 is used as the primary processor and CPU2 is used as the spare processor. In addition, to prevent the interference on the primary or spare processors, we fix the cpu affinity of likwid-powermeter to CPU3.

We setup this experimental as follows:

- We randomly select two benchmarks from the selected set; and assign a period to two benchmarks such that the total utilization of the two benchmarks is equal to an expected utilization;
- We always select the benchmark with smaller utilization as the highly critical task;
- We measure energy consumption of the whole systems within one hyper-period using likwid-powermeter;
- We repeat the experiment for 10 times and compute the average energy consumption for the selected task set.

We consider both criticality aware (CASS) and non-criticality scenarios (SS). We use 'ondemand governor' in Linux as the online approach which dynamically changes CPU frequency in accordance to CPU utilization. The following are the approaches evaluated for comparison in this experiment. Note that in this evaluation we take into account the Intel turbo boost [34] which is very similar to a race-to-idle approach [35]:

Table 10
Experimental results on the real platform.

Utilization	0.6		0.7		0.8		0.9	
Benchmarks	blackscholes & vips		freqmine & facesim		bodytrack & blackscholes		streamcluster & x264	
Approach	SS	CASS	SS	CASS	SS	CASS	SS	CASS
ondemand w tb	995.6J	993.5J	2817.2J	2481.8J	2715.2J	2665.5J	6958.2J	6675.3J
ondemand wo tb	781.9J	777.3J	2420.2J	2391.6J	2133.8J	2058.0J	5162.0J	4763.4J
fixed max freq	802.1J	777.3J	2398.4J	2033.3J	2242.4J	1957.1J	5293.5J	4803.9J
CASS+		695.368		1782.7J		1738.2J		4392.3J

- ondemand without turbo boost: CPU frequency changes according to the CPU workload [28], and in this setting, turbo boost feature is off;
- ondemand with turbo boost: also use 'ondemand' policy but turbo boost feature is on;
- fixed max frequency: in this setting, we fix the operational frequency to the maximum frequency;
- CASS+ frequency: we use CASS+ algorithm to find a fixed frequency for the system.

All the experimental results are summarized in Table 10, where four groups of benchmarks are formed, representing four different utilizations., 0.6, 0.7, 0.8, 0.9. From the experimental results, we observe the following:

- As reported in [34], Turbo boost is not an energy-efficient policy. For the SS technique, it shows the same trend. This may indicate that the race-to-idle approach is not a good way to achieve energy efficiency in SS framework. It is seen that in the worst case of streamcluster & x264 with turboboost feature consumes more energy consumption by 28%;
- The online approach does not lead to energy efficiency even in comparison with 'fixed max frequency'. In freqmine & facesim and bodytrack & blackscholes, the online approach even consumes more enegy than the maximum frequency. This can serve as an evidence that in some cases the dynamic approach may be not a good choice for periodic real-time systems to achieve energy efficiency;
- CASS+ finds an energy-efficient frequency for all cases. In the best case of bodytrack & blackscholes, CASS+ can save energy consumption by 15%.

From the real platform results, we find that the cluster system can be used to provide an energy-efficient platform for the SS techniqe.

7. Conclusions

In this paper, inspired by mixed-criticality systems, we present the CASS framework which integrates the concept of criticalities into the SS framework. The novel CASS framework can achieve more energy efficiency by trading off the fault tolerance of some lower critical tasks. In addition, in contrast to the existing SS techniques which use an online approach to determine the energy-efficient frequency for the primary processor, we propose an offline approach to determine the operational frequency. This offline approach can mitigate the overhead due to the frequent scaling. In addition, we consider the cluster systems for the SS framework and present an algorithm to determine the operational frequency for the cluster systems. We use a case study, synthetic task sets and real benchmarks on a real-platform to evaluate our proposed approaches.

Reliability management of real-time systems has attracted increasing attention, and in this paper we introduce the criticality concept to the SS framework. This work is the first attempt to investigate the trade-off between the fault tolerance ability of low critical tasks and energy efficiency the whole system. The future work can be done in the combination of our proposed offline approach and the existing online approach to achieve the ultimate energy efficiency. Moreover, the heterogeneous systems are gradually replacing homogeneous systems as the major computing systems, so it would be an interesting problem to investigate how to implement the CASS on the heterogeneous systems, like [36].

Declaration of Competing Interest

None

Acknowledgment

This work is partially supported by NSFC 61902341 and 61801418.

References

[1] R.C. Baumann, Radiation-induced soft errors in advanced semiconductor technologies, IEEE Trans. Device Mater. Reliab. 5 (3) (2005) 305–316.
[2] A. Burns, R. Davis, S. Punnekkat, Feasibility analysis of fault-tolerant real-time task sets, in: Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, 1996, pp. 29–33, doi:10.1109/EMWRTS.1996.557785.
[3] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, Distributed fault-tolerant real-time systems: the mars approach, IEEE Micro 9 (1) (1989) 25–40, doi:10.1109/40.16792.
[4] D. Zhu, H. Aydin, Energy management for real-time embedded systems with reliability requirements, in: Proceedings of the ICCAD, 2006, pp. 528–534.
[5] M.A. Haque, H. Aydin, D. Zhu, Energy-aware standby-sparing technique for periodic real-time applications, in: Proceedings of the ICCD, 2011, pp. 190–197.
[6] A. Ejlali, B.M. Al-Hashimi, P. Eles, Low-energy standby-sparing for hard real-time systems, IEEE TCAD 31 (3) (2012) 329–342.
[7] J.W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung, Imprecise computations, Proc. IEEE 82 (1) (1994).
[8] G.C. Buttazzo, G. Lipari, M. Caccamo, L. Abeni, Elastic scheduling for flexible workload management, IEEE Trans. Comput. 51 (3) (2002).
[9] A. Burns, R. Davis, Mixed Criticality Systems-A Review, University of York (2015).
[10] P. Huang, H. Yang, L. Thiele, On the scheduling of fault-tolerant mixed-criticality systems, in: Proceedings of the DAC, 2014, pp. 1–6.
[11] S. Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS), 2007.
[12] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S.V. der Ster, L. Stougie, The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems, in: Proceedings of the 24th Euromicro Conference on Real-Time Systems, 2012.
[13] G. v. d. Bruggen, K. Chen, W. Huang, J. Chen, Systems with dynamic real-time guarantees in uncertain and faulty execution environments, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), 2016, pp. 303–314, doi:10.1109/RTSS.2016.037.
[14] S. Herbert, D. Marculescu, Analysis of dynamic voltage/frequency scaling in chip-multiprocessors, in: Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '07), 2007, pp. 38–43, doi:10.1145/1283780.1283790.
[15] S. Pagani, et al., Energy efficiency for clustered heterogeneous multicores, IEEE TPDS 28 (5) (2017) 1315–1330.
[16] C. Bienia, S. Kumar, J.P. Singh, K. Li, The parsec benchmark suite: Characterization and architectural implications, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, in: PACT '08, ACM, New York, NY, USA, 2008, pp. 72–81, doi:10.1145/1454115.1454128.
[17] K. Jeffay, D.F. Stanat, C.U. Martel, On non-preemptive scheduling of period and sporadic tasks, in: Proceedings of the Twelfth Real-Time Systems Symposium, 1991, pp. 129–139, doi:10.1109/REAL.1991.160366.
[18] M.A. Haque, H. Aydin, D. Zhu, Energy management of standby-sparing systems for fixed-priority real-time workloads, in: Proceedings of the IGCC, 2013, pp. 1–10.
[19] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM (JACM) (1973).
[20] A. Mazouz, et al., Evaluation of cpu frequency transition latency, Comput. Sci. Res. Devel. 29 (3) (2014) 187–195.

[21] Q. Zhao, Z. Gu, H. Zeng, N. Zheng, Schedulability analysis and stack size minimization with preemption thresholds and mixed-criticality scheduling, J. Syst. Arch. Embedded Syst. Des. 83 (2018) 57–74, doi:10.1016/j.sysarc.2017.03.007.
[22] M. Bambagini, et al., Energy-aware scheduling for real-time systems: a survey, ACM TECS 15 (1) (2016) 7:1–7:34, doi:10.1145/2808231.
[23] R. Melhem, D. Mosse, E. Elnozahy, The interplay of power management and fault recovery in real-time systems, IEEE Trans. Comput. 53 (2) (2004) 217–231, doi:10.1109/TC.2004.1261830.
[24] H. Chetto, M. Chetto, Some results of the earliest deadline scheduling algorithm, IEEE TSE 15 (10) (1989) 1261–1269.
[25] J. Goossens, R. Devillers, Feasibility intervals for the deadline driven scheduler with arbitrary deadlines, in: Proceedings of the RTCSA, 1999, pp. 54–61.
[26] D. Liu, et al., Energy-efficient scheduling of real-time tasks on heterogeneous multicores using task splitting, in: Proceedings of the RTCSA, 2016, pp. 149–158.
[27] J. Treibig, G. Hager, G. Wellein, Likwid: A lightweight performance-oriented tool suite for x86 multicore environments, in: Proceedings of the 39th International Conference on Parallel Processing Workshops, 2010, pp. 207–216, doi:10.1109/ICPPW.2010.38.
[28] V. Pallipadi, A. Starikovskiy, The ondemand governor, in: Proceedings of the Linux Symposium, 2, 2006.
[29] P. Pillai, K.G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, SIGOPS Oper. Syst. Rev. 35 (5) (2001) 89–102, doi:10.1145/502059.502044.
[30] A. Burns, F. Zhang, Schedulability analysis for real-time systems with EDF scheduling, IEEE Trans. Comput. 58 (2009) 1250–1258, doi:10.1109/TC.2009.58.
[31] S.K. Baruah, A.K. Mok, L.E. Rosier, Preemptively scheduling hard-real-time sporadic tasks on one processor, in: Proceedings of the RTSS, 1990, pp. 182–190.
[32] P. Huang, et al., Service adaptions for mixed-criticality systems, in: Proceedings of the ASP-DAC, 2014, pp. 125–130.
[33] E. Bini, G.C. Buttazzo, Measuring the performance of schedulability tests, Real-Time Syst. 30 (1–2) (2005) 129–154.
[34] J. Charles, P. Jassi, N.S. Ananth, A. Sadat, A. Fedorova, Evaluation of the intel core i7 turbo boost feature, in: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 188–197, doi:10.1109/IISWC.2009.5306782.
[35] H. Hoffmann, Racing and pacing to idle: An evaluation of heuristics for energy-aware resource allocation, in: Proceedings of the Workshop on Power-Aware Computing and Systems, in: HotPower '13, ACM, New York, NY, USA, 2013, pp. 13:1–13:5, doi:10.1145/2525526.2525854.
[36] A. Roy, H. Aydin, D. Zhu, Energy-aware standby-sparing on heterogeneous multicore systems, in: Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC), 2017, pp. 1–6, doi:10.1145/3061639.3062238.

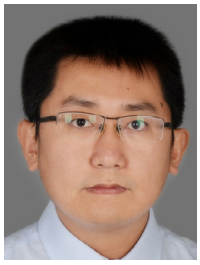
Mingxiong Zhao received the B.S. degree and the Ph.D. degree from South China University of Technology (SCUT), Guangzhou, China, in 2011 and 2016, respectively. Since 2016, he has been an Assistant Professor at the School of Software, Yunnan University, Kunming, China. His current research interests are embedded systems, physical layer security, cooperative relay communication, and social aware communication systems.

Di Liu received the B.Eng. and M.Eng. degrees from Northwestern Polytechnical University, China, in 2007 and 2011, respectively, and the Ph.D. degree from Leiden University, The Netherlands, in 2017. He is currently an Assistant Professor with the School of Software, Yunnan University, China. His research interests include the fields of real-time systems, energy-efficient multicore/many core systems, and cyber-physical systems.

Xu Jiang received the B.S. degree from Northwestern Polytechnical University, Xi'an, China, in 2009, the M.S. degree from the Graduate School of the Second Research Institute, China Aerospace Science and Industry Corporation, Beijing, China, in 2012, and the Ph.D. degree from Beihang University, China, in 2018. Currently, he is an Assistant Professor with the School of Computer Science and Engineering, University of Electronical Science and Technology of China, China. His current research interests include real-time systems, parallel and distributed systems, and embedded systems.

Weichen Liu received the B.Eng. and M.Eng. degrees from the Harbin Institute of Technology, Harbin, China, and the Ph.D. degree from the Hong Kong University of Science and Technology, Hong Kong. He is an Assistant Professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His current research interests include embedded and real-time systems, multiprocessor systems, and fault-tolerant systems.

Gang Xue received the B.S. degree from Wuhan Technical University of Surveying and Mapping in 2000 and the M.Sc. degree and the Ph.D. degree from Yunnan University in 2006 and 2009, respectively. He is currently an Associate Professor with the School of Software, Yunnan University, China. His interests are embedded systems, time-sensitive computing, service computing.



Cheng Xie received the B.S. degree in software engineering from the Minzu University of China, Beijing, China, in 2009, and the M.S. and Ph.D. degrees in software engineering from Shanghai Jiao Tong University, Shanghai, China, in 2012 and 2017, respectively. He is currently an Assistant Professor with the School of Software, Yunnan University, Kunming, China. His research interests include semantic web, linked open data, knowledge graph, and ontology.

Yun Yang received the B.Sc. degree (Hons.) in information technology and telecommunication from Lancaster University, Lancaster, U.K., in 2004, the M.Sc. degree in advanced computing from Bristol University, Bristol, U.K., in 2005, and the M.Phil. degree and the Ph.D. degree from The University of Manchester in 2006 and 2011, respectively. He is currently a Full Professor with the School of Software, Yunnan University, Kunming, China. His current research interests include machine learning, data mining, pattern recognition, and temporal data process and analysis.

Zhishan Guo received the B.E. degree from Tsinghua University, Beijing, China, in 2009, the M.Phil. degree from the Chinese University of Hong Kong, Hong Kong, in 2011, and the Ph.D. degree from the University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, in 2016. He is an Assistant Professor with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL, USA. His current research interests include real-time scheduling, cyber-physical systems, and neural networks and their applications.